

Relatively Complete Counterexamples for Higher-Order Programs

Phúc C. Nguyễn

University of Maryland, USA
pcn@cs.umd.edu

David Van Horn

University of Maryland, USA
dvanhorn@cs.umd.edu



Abstract

In this paper, we study the problem of generating inputs to a higher-order program causing it to error. We first approach the problem in the setting of PCF, a typed, core functional language and contribute the first relatively complete method for constructing counterexamples for PCF programs. The method is relatively complete with respect to a first-order solver over the base types of PCF. In practice, this means an SMT solver can be used for the effective, automated generation of higher-order counterexamples for a large class of programs.

We achieve this result by employing a novel form of symbolic execution for higher-order programs. The remarkable aspect of this symbolic execution is that even though symbolic higher-order inputs and values are considered, the path condition remains a first-order formula. Our handling of symbolic function application enables the reconstruction of higher-order counterexamples from this first-order formula.

After establishing our main theoretical results, we sketch how to apply the approach to untyped, higher-order, stateful languages with first-class contracts and show how counterexample generation can be used to detect contract violations in this setting. To validate our approach, we implement a tool generating counterexamples for erroneous modules written in Racket.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Software/Program Verification; D.3.1 [Programming Languages]: Formal Definitions and Theory

General Terms Verification

Keywords Higher-order programs; symbolic execution; contracts

1. Introduction

Generating inputs that crash first-order programs is a well-studied problem in the literature on symbolic execution [Cadar et al. 2006; Godefroid et al. 2005], type systems [Foster et al. 2002], flow analysis [Xie and Aiken 2005], and software model checking [Yang et al. 2004]. However, in the setting of higher-order languages, those that treat computations as first-class values, research has largely focused on the verification of programs without investigating how to effectively report counterexamples as concrete inputs

when verification fails (e.g., Rondon et al. [2008]; Xu et al. [2009]; Kawaguchi et al. [2010]; Vytiniotis et al. [2013]; Tobin-Hochstadt and Van Horn [2012]; Nguyễn et al. [2014]).

There are, however, a few notable exceptions which tackle the problem of counterexamples for higher-order programs. Perhaps the most successful has been the approach of random testing found in tools such as *QuickCheck* [Claessen and Hughes 2000; Klein et al. 2010]. While testing works well, it is not a complete method and often fails to generate inputs for which a little symbolic reasoning could go further. Symbolic execution aims to overcome this hurdle, but previous approaches to higher-order symbolic execution can only generate *symbolic* inputs, which are not only less useful to programmers, but may represent infeasible paths in the program execution [Tobin-Hochstadt and Van Horn 2012; Nguyễn et al. 2014]. Higher-order model checking [Kobayashi 2013] offers a complete decision procedure for typed, higher-order programs with finite base types, and can generate inputs for programs with potential errors. Unfortunately, only first-order inputs are allowed. This assumption is reasonable for whole programs, but not suitable for testing higher-order *components*, which often consume and produce behavioral values (e.g., functions, objects). Zhu and Jagannathan [2013] give an approach to dependent type inference for ML that relies on counterexample refinement. This approach can be used to generate higher-order counterexamples, however no measure of completeness is considered.

In this paper, we solve the problem of generating potentially higher-order inputs to functional programs. We give the first relatively complete approach to generating counterexamples for PCF programs. Our approach uses a novel form of symbolic execution for PCF that accumulates a path condition as a symbolic heap. The semantics is an adaptation of Nguyễn et al. [2014], where the critical technical distinction is our semantics maintains a *complete* path condition during execution. The key insight of this work is that although the space of higher-order values is huge, it is only necessary to search for counterexamples from a subset of functions of specific shapes. Symbolic function application can be leveraged to decompose unknown functions to lower-order unknown values. By the point at which an error is witnessed, there are sufficient *first-order* constraints to reconstruct the potentially higher-order inputs needed to crash the program. The completeness of generating counterexamples reduces to the completeness of solving this first-order constraint, and in this way is relatively complete [Cook 1978].

Beyond PCF, we show the technique is not dependent on assumptions of the core PCF model such as type safety and purity. We sketch how the approach scales to handle untyped, higher-order, imperative programs. We also show the approach seamlessly scales to handle first-class behavioral contracts [Findler and Felleisen 2002] by incorporating existing semantics for contract monitoring [Dimoulas et al. 2012] with no further work. The semantic decomposition of higher-order contracts into lower-order functions

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PLDI'15, June 13–17, 2015, Portland, OR, USA.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 978-1-4503-3468-6/15/06...\$15.00.

<http://dx.doi.org/10.1145/2737924.2737971>

naturally composes with our model of unknown functions to yield a contract counterexample generator for Contract PCF (CPCF).

Contributions We make the following contributions:

1. We give a novel symbolic execution semantics for PCF that gradually refines unknown values and maintains a complete path condition.
2. We give a method of integrating a first-order solver to simultaneously obtain a precise execution of symbolic programs and enable the construction of higher-order counterexamples in case of errors.
3. We prove our method of finding counterexamples is sound and relatively complete.
4. We discuss extensions to our method to handle untyped, higher-order, imperative programs with contracts.
5. We implement our approach as an improvement to a previous contract verification system, distinguishing definite program errors from potentially false positives.

Outline The remainder of the paper is organized as follows. We first step through a worked example of a higher-order program that consumes functional inputs (§ 2). Stepping through the example illustrates the key ideas of how the path condition is accumulated as a heap of potentially symbolic values with refinements and how this heap can be translated to a first-order formula suitable for an SMT solver. Generating a model for the path condition at the point of an error reconstructs the higher-order input needed to witness the error. Next, we develop the core model of Symbolic PCF (§ 3) as a heap-based reduction semantics. We prove that the semantics is sound and relatively complete, our main theoretical contribution. We then show how to scale the approach beyond PCF to untyped, higher-order, imperative languages with contracts (§ 4). We use these extensions as the basis of a tool for finding contract violations in Racket code to validate our approach (§ 5). Finally, we relate our work to the literature (§ 6) and conclude (§ 7).

2. Worked examples

We illustrate our idea using an *incomplete* OCaml program. The basic idea is that we give a semantics to incomplete programs using a heap of refinements that constrain all possible completions of the program. When an error is reached, the heap is given to an SMT solver, which constructs a model that represents a counterexample.

As our example we use a function f that takes as its arguments a function g and a number n and performs a division whose denominator involves the application of g to n . We write \bullet^T to denote an unknown value of the appropriate type (and omit the type when it is clear from context). This example, though contrived, is small and conveys the heart of our method.

```
let f (g : int → int) (n : int) : int =
  1 / (100 - (g n))
in
(• f)
```

Now let us consider the possible errors that can arise from running this code for any interpretation of the unknown value.

Although the application of unknown function \bullet is an arbitrary computation that can result in any error, we restrict our attention to possible errors stemming from misbehavior of the visible part of the above code and assume function \bullet is bug-free. Through symbolic execution and incremental refinement of unknown values, we reveal one implementation of \bullet that triggers a division error in f 's implementation.

```
Error: Division_by_zero
Breaking context:
```

$\bullet = \text{fun } f \rightarrow (f (\text{fun } n \rightarrow 100)) \ 0)$

To find a counterexample, we first seek a possible error by running the program under an extended reduction semantics allowing *unknown*, or *opaque*, values. When execution follows different branches, it remembers assumptions associated with each path, and opaque values become partially *known*, or *transparent*. To keep track of incremental refinements throughout execution, we allocate all values in a heap and maintain an upper bound to the behavior of each unknown value.

The semantics takes the form of a reduction relation on pairs of expressions and heaps, written $\langle E, \Sigma \rangle \mapsto \langle E', \Sigma' \rangle$. In our example, the first step of computation is to allocate a fresh location to hold the unknown function being applied.

$$\langle (\bullet \ f), \emptyset \rangle \mapsto \langle (L_1 \ f), [L_1 \mapsto \bullet] \rangle$$

Allocating values in the heap this way gives us a means to refine values and to communicate these refinements to later parts of the computation.

At this point, we can partially solve for the unknown value. Since it is applied to f , it must be a function of one argument. But how can we solve for the body of the function? The key observation is that while many possible solutions for the function body may exist, if the function can reach an error state, then it can reach that error state by immediately applying the input to some arguments, *without loss of generality*. Since the input function takes two arguments, we can partially solve for the body of the function as “apply the input to two unknown values.” By allocating these two unknowns and refining f , we arrive at the state:

$$\begin{aligned} \langle (f \ L_2 \ L_3), [L_1 \mapsto \text{fun } f \rightarrow (f \ L_2 \ L_3), \\ L_2 \mapsto \bullet^{\text{int} \rightarrow \text{int}}, \\ L_3 \mapsto \bullet^{\text{int}}] \rangle \end{aligned}$$

The program then executes f 's body, substituting g with L_2 and n with L_3 . The next sub-expression to reduce is $(g \ n)$, which is $(L_2 \ L_3)$ after substitution, which is yet another unknown function application, so the next step is to partially solve for L_2 . Unlike in the higher-order case, there is no interaction with the input value that needs to be considered (since it is not behavioral), so the function can simply return a new, unknown output, L_4 , giving us the following transition:

$$\begin{aligned} \langle (L_2 \ L_3), [L_1 \mapsto \text{fun } f \rightarrow (f \ L_2 \ L_3), \\ L_2 \mapsto \bullet^{\text{int} \rightarrow \text{int}}, \\ L_3 \mapsto \bullet^{\text{int}}] \rangle \\ \mapsto \langle (L_4, [L_1 \mapsto \text{fun } f \rightarrow (f \ L_2 \ L_3), \\ L_2 \mapsto \text{fun } n \rightarrow L_4, \\ L_3 \mapsto \bullet^{\text{int}}, \\ L_4 \mapsto \bullet^{\text{int}}]) \rangle \end{aligned}$$

At this point, we need to compute $100 - L_4$, i.e. subtract an unknown integer from 100. The solution is simple, we extend the primitive arithmetic operations to produce new unknown values and annotate the unknown result with a predicate to embed the knowledge that it is equal to $100 - L_4$:

$$\begin{aligned} \langle 100 - L_4, [L_1 \mapsto \text{fun } f \rightarrow (f \ L_2 \ L_3), \\ L_2 \mapsto \text{fun } n \rightarrow L_4, \\ L_3 \mapsto \bullet^{\text{int}}, \\ L_4 \mapsto \bullet^{\text{int}}] \rangle \\ \mapsto \langle L_5, [L_1 \mapsto \text{fun } f \rightarrow (f \ L_2 \ L_3), \\ L_2 \mapsto \text{fun } n \rightarrow L_4, \\ L_3 \mapsto \bullet^{\text{int}}, \\ L_4 \mapsto \bullet^{\text{int}}, \\ L_5 \mapsto \bullet^{\text{int}}, \text{fun } x \mapsto x = (100 - L_4)] \rangle \end{aligned}$$

We finally arrive at the point of computing $1 / L_5$. At this point the semantics branches non-deterministically since L_5 may represent a zero or non-zero value. In the case of an error, we refine L_5 to be zero, giving us the final state:

```
(error, [L1 ↦ fun f → (f L2 L3),
        L2 ↦ fun n → L4,
        L3 ↦ •int,
        L4 ↦ •int,
        L5 ↦ •int, fun x → x=0, fun x → x=(100-L4)])
```

At this point, the program has reached an error state and has accumulated a heap of invariants that constrain the unknown values. But notice that since functions have been partially solved for as they've been applied, there are only first-order unknowns in the heap. At this point, translation of refinements on integers into first-order assertions is straightforward:

```
(declare-const L3 Int)
(declare-const L4 Int)
(declare-const L5 Int)
(assert (= L5 (- 100 L4)))
(assert (= 0 L5))
```

A solver such as Z3 [De Moura and Bjørner 2008] can easily solve such constraints and yield ($L_3 = 0$, $L_4 = 100$, $L_5 = 0$) as a model. We then plug these values into the current heap and straightforwardly obtain the counterexample shown at the start.

In summary, we use execution to incrementally construct the shape of each function, query a first-order solver for a model for base values, and combine these first-order values to construct a higher-order counterexample.

3. Formal model with Symbolic PCF

This section presents a reduction semantics illustrating the core of our approach. Symbolic PCF (SPCF) [Tobin-Hochstadt and Van Horn 2012] extends the PCF language [Scott 1993] with *incomplete* programs containing *symbolic values* that can be higher-order.

We present the language's syntax and semantics, describe its integration with an external solver, and show how the semantics enables the generation of a counterexample when an error occurs. Finally, we prove that our counterexample construction is sound and complete relative to the underlying solver. The key technical challenge in designing such a semantics is to make sure not to over-constrain unknowns, which would be unsound, while also not under-constraining unknowns, which would be incomplete.

3.1 Syntax of SPCF

Figure 1 presents the syntax of SPCF. We write \vec{E} to mean a sequence of expressions and treat it as a set where convenient. The language is simply typed with typical expression forms for conditionals, applications, primitive applications, recursion, and values such as natural numbers and lambdas. The evaluation context \mathcal{E} is standard for a call-by-value semantics. We highlight non-standard forms in gray. The key extension of SPCF compared to PCF is the notion of *symbolic*, or *opaque* values. We write \bullet^T to mean an unknown but fixed and syntactically closed value¹ of type T . The system automatically annotates each opaque value with a unique label to identify its source location. It also uniquely labels each source location that could have a potential run-time failure. In SPCF, such failures can only occur with the application of partial, primitive operations.

When evaluating an SPCF expression, we allocate all values and maintain a heap to keep track of their constraints. When execution proceeds through conditional branches and primitive op-

Expressions E	$::=$	$A \mid V \mid X \mid \text{if } EEE \mid EE \mid O \vec{E}^L$
Contexts \mathcal{E}	$::=$	$[\] \mid \text{if } \mathcal{E} EE \mid \mathcal{E} E \mid L \mathcal{E} \mid O \vec{L} \vec{E} \vec{E}$
Values V	$::=$	$\bullet^T_L \mid \lambda X : T. E \mid n$
Answers A	$::=$	$L^T \mid \text{err}_O^L$
Operations O	$::=$	$\text{zero?} \mid \text{add1} \mid \text{div1} \mid \dots$
Predicates P	$::=$	$\lambda X : T. E$
Types T	$::=$	$\text{nat} \mid T \rightarrow T$
Heaps Σ	$::=$	$\emptyset \mid \Sigma, L \mapsto S$
Storeables S	$::=$	$\bullet^{\{T \vec{P}\}} \mid \lambda X : T. E \mid n \mid \text{case}^T \vec{L} \mapsto \vec{L}$
Variables $X, L \in$		<i>identifier</i>

Figure 1. Syntax of SPCF

erations, we refine the heap at appropriate locations with stronger assumptions taken at each branch. As figure 1 shows, a heap is a finite function mapping each location L to a stored value S as an upper bound of the value's run-time behavior. A stored value S is similar to a syntactic value, but a stored unknown value can be further refined by arbitrary program predicates. For example, $\bullet^{\{\text{nat}, \lambda x. \text{even? } x\}}$ denotes an *unknown* even natural number.

In addition, we use $\text{case}^T [\vec{L} \mapsto \vec{L}]$ to denote a mapping approximating an unknown function of type $\text{nat} \rightarrow T$. We clarify the role of this construct later when discussing the semantics of applying opaque functions, but the intuition is that this form is used to constrain unknown functions (of base type input) to always produce the same result when given the same input; it is critical for achieving completeness and is not present in the original SPCF semantics of Tobin-Hochstadt and Van Horn [2012].

Syntax for answers A is internal and unavailable to programmers. An answer is either a location L^T pointing to a value of type T on the heap, or an error message err_O^L *blaming* source location L for violating primitive O 's precondition. A source location in an error message is not just for precise blaming, but is important in defining what it means to have a sound symbolic execution, as we will discuss in detail in section 3.2.

We omit straightforward type-checking rules for SPCF and assume all considered programs are well-typed. In addition, we omit showing types and labels for constructs such as locations and lambdas when they are irrelevant or clear from context.

In the following, we use the term *unknown program portion* to refer to all unknown values in the original (incomplete) program, and *known program portion* to refer to the rest of it.

3.2 Semantics of SPCF

We present the semantics of SPCF as a relation between states of the form $\langle E, \Sigma \rangle$. Key extensions to the straightforward concrete semantics include generalization of primitives to operate on symbolic values and reduction rules for opaque applications. Intuitively, reduction on abstract states approximates reduction on concrete states, accounting for all possible instantiations of symbolic values. Figure 2 presents the reduction semantics of SPCF. The semantics is also mechanized as a Redex model and available online.²

Each value is allocated on the heap and reduces to a location as shown in rules *Opq* and *Conc*. Because an opaque value stands for an arbitrary but fixed and closed value, we reuse a location if it has been previously allocated.

Rule *Prim* shows the reduction of a primitive application. We use δ to relate primitive operators and values to results. Typically,

¹For example, \bullet does not approximate $(\lambda x. y)$

²<https://github.com/philnguyen/soft-contract/tree/pldi-2015/soft-contract/ce-redex>

δ is a function, but here it is a relation because primitive operations may behave non-deterministically on unknown values. In addition, the relation includes a heap to remember assumptions in each taken branch. Rules for conditionals are straightforward, except we also rely on δ to determine the truth of the value branched on instead of replicating the logic. We use \emptyset to indicate falsehood and any non-zero number for truth (as in PCF). Application of a λ -abstraction follows standard β -reduction.

Application of an unknown value to a function argument results in a range of possibilities to consider. This space, however, can be partitioned into a few cases. First, the unknown program portion can have bugs of its own regardless of the argument, but our concern is only to find bugs in the known program portion so the possibility of these errors is ignored. Second, the function argument escapes into an unknown context and can be invoked in an arbitrary way. However, any invocation triggering an error can be reduced to a chain of function applications. Alternatively, the unknown function may not explore its argument's behavior directly during the execution of its body, but delay that in a returned closure. Finally, the unknown function may completely ignore its argument and fail to reveal any hidden bug, allowing the program to proceed to other parts. These four cases result in specific shapes a function can have. Therefore, upon opaque function application, we refine the opaque function's shape accordingly.

Consider this example:

$$(\bullet_{L_1}^{(\text{nat} \rightarrow \text{nat}) \rightarrow T}) (\lambda x : \text{nat}. (/ \ 1 \ x)^L)$$

and the following possible instantiations of L_1 :

1. $\lambda f. (/ \ 1 \ \emptyset)$
2. $\lambda f. (f \ \emptyset)$
3. $\lambda f. \lambda x. (\text{add1 } (f \ x))$
4. $\lambda f. \lambda x. 42$

Completion (1) raises an error from within the unknown function blaming L_1 itself, (2) triggers the division error blaming L , (3) delays the exploration of its argument's behavior by returning a closure referencing the argument, and (4) is a constant function ignoring its argument. As we are only interested in errors in the known program portion, we ignore behavior such as (1). Rules *AppOpq1*, *AppOpq2*, *AppOpq3* and *AppHavoc* model the remaining possibilities.

Rule *AppOpq1* shows a simple case where the argument is a first-order value with no behavior. In this case, we approximate the application's result with a symbolic value of appropriate type, and refine the opaque function to be of the form $\text{case}^T [L \mapsto \tilde{L}]$ to remember this mapping. Any future application of this function to an equal argument gives an equal result.

Applying a higher-order opaque function results in multiple distinct possibilities. Rule *AppOpq2* considers the case where the function ignores its argument (i.e. it is a constant function). Any future application of this unknown function gives the same result. Rule *AppOpq3* considers the case where the unknown context does not immediately explore its argument's behavior but delays that work by wrapping the argument inside another function. The context using this result may or may not reveal a potential error. Finally, rule *AppHavoc* considers the case where the unknown context explores its argument's behavior by supplying an unknown value to its argument and putting the result back into another unknown context.

When the argument is higher-order, we do not use a simple dispatch as in rule *AppOpq1* because there is no mechanism for comparing functions for equality (without applying them as in rule *AppHavoc*).

Application rules for mappings are straightforward. Rule *AppCase1* reuses the result's location for a previously seen application, whereas rule *AppCase2* allocates a fresh location for the result of a newly seen application.

$$\begin{aligned} \delta(\Sigma, \text{zero?}, L) &\ni \langle 1, \Sigma \rangle && \text{if } \Sigma \vdash L : \text{zero?} \checkmark \\ \delta(\Sigma, \text{zero?}, L) &\ni \langle \emptyset, \Sigma \rangle && \text{if } \Sigma \vdash L : \text{zero?} \times \\ \delta(\Sigma, \text{zero?}, L) &\supseteq \{ \langle 1, \Sigma[L \mapsto \emptyset] \rangle, \langle \emptyset, \Sigma[L \mapsto \neg \text{zero?}] \rangle \} && \text{if } \Sigma \vdash L : \text{zero?} ? \\ \delta(\Sigma, \text{div}, L_1, L_2) &\ni \langle m/n, \Sigma \rangle && \text{if } \Sigma(L_1) = m \text{ and } \Sigma(L_2) = n, n \neq \emptyset \\ \delta(\Sigma, \text{div}, L_1, L_2) &\ni \langle \bullet^{\text{nat}}, (\equiv L_1 / L_2), \Sigma' \rangle && \text{if } \Sigma(L_2) \neq n \text{ and } \delta(\Sigma, \text{zero?}, L_2) \ni \langle \emptyset, \Sigma' \rangle \\ \delta(\Sigma, \text{div}, L_1, L_2) &\ni \langle \text{err}_{\text{div}}, \Sigma' \rangle && \text{if } \Sigma(L_2) \neq n \text{ and } \delta(\Sigma, \text{zero?}, L_2) \ni \langle 1, \Sigma' \rangle \end{aligned}$$

Figure 3. Selected Primitive Operations

These rules for opaque application collectively model the *de-mononic* context in previous work on higher-order symbolic execution [Tobin-Hochstadt and Van Horn 2012], but they unroll the unknown context incrementally and remember its shape to enable counterexample construction when execution finishes.

Finally, we define the semantics to be the contextual closure of all the above reductions (rule *Close*). Errors halt the program and discard the context (rule *Error*).

3.3 Primitive operations

We rely on relation δ to interpret primitive operations. The rules straightforwardly extend standard operators to work on symbolic values. In particular, division by an unknown denominator non-deterministically either returns another integer or raises an error. The relation also remembers appropriate refinements to arguments and results at each branch. Figure 3 presents a selection of representative rules for primitive operations *zero?* and *div*. We abbreviate $\lambda X. (=X E)$ as $(\equiv E)$. Rules for primitive predicates such as *zero?* utilize a proof relation between the heap, the value, and a predicate, which we present next.

3.4 Proof relation

We define a proof relation deciding whether a value satisfies a predicate. We write $\Sigma \vdash L : P \checkmark$ to mean the value at location L definitely satisfies predicate P , which implies that all possible instantiations of L satisfy P . In the same way, $\Sigma \vdash L : P \times$ means all instantiations of L definitely fail P . Finally, $\Sigma \vdash L : P ?$ is a conservative answer when we cannot draw a conclusion given information from existing refinements on the heap.

Precision of our execution relies on this proof relation. (A trivial relation answering “neither” for all queries would make the execution sound though highly imprecise.) Instead of implementing our own proof system, we rely on an SMT solver for sophisticated reasoning on base values.

Figure 4 shows the translation $\{\cdot\}$ of run-time constructs into logical formulas. The translation of a heap is the conjunction of formulas obtained from each location and its value, and the translation of each location and value is straightforward. In particular, a location pointing to a concrete number translates to the obvious assertion of equality, and a mapping (case $\tilde{L} \mapsto \tilde{L}$) adds constraints asserting that equal inputs imply equal outputs. Since outputs of maps may be functions, it might appear as though we need function equality. However, we do not need general equality on functions, but just a specialized equality that can handle those opaque functions generated by *AppOpq1*, *AppOpq2*, *AppOpq3* and *AppHavoc*. Equality on similar function forms proceeds structurally, while equality on different function forms translate trivially to *False* (not shown).

Notice that the proof system only needs to handle predicates of simple forms and not their arbitrary compositions. We rely

$\langle \bullet_L^T, \Sigma \rangle \mapsto \langle L^T, \Sigma' \rangle$	where $\Sigma' = \Sigma[L \mapsto \bullet_L^T]$ if $L \notin \text{dom}(\Sigma)$, or Σ otherwise	[Opq]
$\langle V, \Sigma \rangle \mapsto \langle L, \Sigma[L \mapsto V] \rangle$	where $L \notin \text{dom}(\Sigma)$ and $V \neq \bullet$	[Conc]
$\langle \text{if } L \ E_1 \ E_2, \Sigma \rangle \mapsto \langle E_1, \Sigma' \rangle$	$\delta(\Sigma, \text{zero?}, L) \ni \langle 0, \Sigma' \rangle$	[IfTrue]
$\langle \text{if } L \ E_1 \ E_2, \Sigma \rangle \mapsto \langle E_2, \Sigma' \rangle$	$\delta(\Sigma, \text{zero?}, L) \ni \langle 1, \Sigma' \rangle$	[IfFalse]
$\langle \langle O \ \vec{L} \rangle, \Sigma \rangle \mapsto \langle L', \Sigma'[L' \mapsto V] \rangle$	if $\delta(\Sigma, O, \vec{L}) \ni \langle V, \Sigma' \rangle$ and $L' \notin \text{dom}(\Sigma')$	[Prim]
$\langle \langle L \ L_x \rangle, \Sigma \rangle \mapsto \langle [L_x/X]E, \Sigma \rangle$	if $\Sigma(L) = \lambda X.E$	[AppLam]
$\langle \langle L \ L_x \rangle, \Sigma \rangle \mapsto$ $\langle L_a, \Sigma[L_a \mapsto \bullet^T, L \mapsto \text{case}^T [L_x \mapsto L_a]] \rangle$	if $\Sigma(L) = \bullet^{\text{nat} \rightarrow T}$ and $L_a \notin \text{dom}(\Sigma)$	[AppOpq1]
$\langle \langle L \ L_x \rangle, \Sigma \rangle \mapsto \langle L_a, \Sigma[L_a \mapsto \bullet^T, L \mapsto \lambda x : T'.L_a] \rangle$	if $\Sigma(L) = \bullet^{T' \rightarrow T}, T' = T_1 \rightarrow T_2$ and $L_a \notin \text{dom}(\Sigma)$	[AppOpq2]
$\langle \langle L \ L_x \rangle, \Sigma \rangle \mapsto \langle [L_x/x]V, \Sigma' \rangle$	if $\Sigma(L) = \bullet^{T' \rightarrow T}, T' = T_1 \rightarrow T_2, T = T_3 \rightarrow T_4$ where $\Sigma' = \Sigma[L \mapsto \lambda x : T'.V, L_1 \mapsto \bullet^{T' \rightarrow T}]$ $L_1 \notin \text{dom}(\Sigma)$, and $V = \lambda y : T_3.((L_1 \ x) \ y)$	[AppOpq3]
$\langle \langle L \ L_x \rangle, \Sigma \rangle \mapsto$ $\langle \langle L_2 \ (L_x \ L_1) \rangle, \Sigma[L \mapsto V, L_1 \mapsto \bullet^{T_1}, L_2 \mapsto \bullet^{T_2 \rightarrow T}] \rangle$	if $\Sigma(L) = \bullet^{T' \rightarrow T}, T' = T_1 \rightarrow T_2$, $L_1, L_2, L_a \notin \text{dom}(\Sigma)$, and $V = \lambda x : T'.(L_2 \ (x \ L_1))$	[AppHavoc]
$\langle \langle L \ L_x \rangle, \Sigma \rangle \mapsto \langle L_a, \Sigma \rangle$	if $\Sigma(L) = \text{case} \dots [L_x \mapsto L_a] \dots$	[AppCase1]
$\langle \langle L \ L_x \rangle, \Sigma \rangle \mapsto$ $\langle L_a, \Sigma[L \mapsto \text{case} [L_z \mapsto L_b] \dots [L_x \mapsto L_a]] \rangle$	if $\Sigma(L) = \text{case} [L_z \mapsto L_b] \dots \text{and } L_x \notin \{L_z \dots\}$ and $L_a \notin \text{dom}(\Sigma)$	[AppCase2]
$\langle \mathcal{E}[E], \Sigma \rangle \mapsto \langle \mathcal{E}[E'], \Sigma' \rangle$	if $\langle E, \Sigma \rangle \mapsto \langle E', \Sigma' \rangle$	[Close]
$\langle \mathcal{E}[\text{err}_O^L], \Sigma \rangle \mapsto \langle \text{err}_O^L, \Sigma \rangle$	if $\mathcal{E} \neq []$	[Error]

Figure 2. Semantics of SPCF

on execution itself to break down complex predicates to smaller ones and take care of issues such as divergence and errors in the predicate itself. For example, if the proof system can prove that a value satisfies predicate P , it automatically allows the execution to prove that the value also satisfies $(\lambda x.(\text{or } (P \ x) \ E))$ for an arbitrarily expression E . By the time we have $[L \mapsto \bullet^{\vec{P}}]$, we can assume all predicates \vec{P} have terminated with true on L . Further, because many solvers do not support uninterpreted higher-order functions, we do not assume such a feature, and the translation only produces queries on first-order values. Nevertheless, the symbolic execution itself can reason about higher-order unknown values. Handling higher-order functions on the semantics side and not relying on the theory of uninterpreted functions also potentially allows the method to scale to more realistic language features such as side effects.

For each query between heap Σ , location L and predicate P , we translate known assumptions from the heap to obtain formula ϕ , and the relationship $(L : P)$ to obtain formula ψ . We then consult the solver to obtain an answer. As figure 5 shows, validity of $(\phi \Rightarrow \psi)$ implies that value L definitely satisfies predicate P , and unsatisfiability of $(\phi \wedge \psi)$ means value L definitely refutes P . If neither can be determined, we return the conservative answer.

3.5 Constructing counterexamples

For each answer reached by evaluation, the heap contains refinements to symbolic values in order to reach such results. In particular, refinements on the heap in an error case describe the condition under which the program goes wrong.

Specifically, at the end of evaluation, refinements on the heap are nearly concrete: higher-order symbolic values are broken down into a chain of argument deconstruction and mappings, and first-order symbolic values have precise constraints that identify the execution path. Indeed, a model to the first-order constraints on the heap yields a counterexample to the program. We simply plug first-order concrete values back into the heap.

The reader may wonder if this process always generates an actual counterexample witnessing a real program bug (soundness), and if it always finds counterexample when a bug exists (completeness). The next section clarifies these points.

$$\begin{aligned}
\overrightarrow{\{\{L \mapsto S\}\}} &= \overrightarrow{\{\{L \mapsto S\}\}} \\
\overrightarrow{\{\{L \mapsto n\}\}} &= (L = n) \\
\overrightarrow{\{\{L \mapsto \bullet^{\text{nat}} \vec{P}\}\}} &= \overrightarrow{\{\{L : P\}\}} \\
\overrightarrow{\{\{L \mapsto \text{case} \dots [L_1 \mapsto L_2] \dots [L_3 \mapsto L_4] \dots\}\}} &= (\bigwedge ((L_1 = L_3) \Rightarrow (L_2 = L_4)) \dots) \\
\overrightarrow{\{\{L : (\lambda X.\text{zero? } X)\}\}} &= (L = 0) \\
\overrightarrow{\{\{L : (\lambda X.(= X (+ L_1 L_2)))\}\}} &= (L = (L_1 + L_2)) \\
\overrightarrow{\{\{L_1 = L_2\}\}_{\text{nat}}} &= (L_1 = L_2) \\
\overrightarrow{\{\{L_1 = L_2\}\}_{T \rightarrow T'}} &= \{\{\Sigma(L_1) = \Sigma(L_2)\}\} \\
\overrightarrow{\{\{(\text{case}^T [L_1 \mapsto L_2] \dots \text{case}^T [L_3 \mapsto L_4] \dots)\}\}} &= (\bigwedge (\{\{L_1 = L_3\}\} \Rightarrow \{\{L_2 = L_4\}\}) \dots) \\
\overrightarrow{\{\{\lambda X.L_1 = \lambda X.L_2\}\}} &= \{\{L_1 = L_2\}\} \\
\overrightarrow{\{\{\lambda X.(L_1 (X \ L_2)) = \lambda X.(L_3 (X \ L_4))\}\}} &= (\bigwedge \{\{L_1 = L_3\}\} \wedge \{\{L_2 = L_4\}\}) \\
\overrightarrow{\{\{\lambda X.\lambda Y.((L_1 \ X) \ Y) = \lambda X.\lambda Y.((L_2 \ X) \ Y)\}\}} &= \{\{L_1 = L_2\}\}
\end{aligned}$$

Figure 4. Heap translation

$$\begin{array}{c}
\textit{Proved} \\
\frac{\{\{\Sigma\}\} \Rightarrow \{\{L : P\}\} \text{ is valid}}{\Sigma \vdash L : P \checkmark} \\
\\
\textit{Refuted} \\
\frac{\{\{\Sigma\}\} \wedge \{\{L : P\}\} \text{ is unsat}}{\Sigma \vdash L : P \times} \\
\\
\textit{Ambig} \\
\frac{\{\{\Sigma\}\} \Rightarrow \{\{L : P\}\} \text{ is invalid and } \{\{\Sigma\}\} \wedge \{\{L : P\}\} \text{ is sat}}{\Sigma \vdash L : P ?}
\end{array}$$

Figure 5. Proof rules

3.6 Soundness and completeness of counterexamples

We show that our method of finding counterexamples in a higher-order program is sound and relatively complete. Soundness means that the system only gives an actual counterexample triggering a bug (not a false positive). Relative completeness means that if the program actually contains a bug and the underlying solver can answer all queries on first order data, the system constructs a concrete counterexample witnessing that bug, even when it involves complex interactions between higher-order values.

The statements and proofs of soundness and completeness revolve around a notion of *approximation*, which we first describe before stating our main theorems.

Approximation Relation We define an approximation relation between *concrete* and *abstract* states. A concrete state contains no unknown values, while an abstract state may contain unknowns. We write $\langle E', \Sigma' \rangle \sqsubseteq \langle E, \Sigma \rangle$ to mean “ $\langle E, \Sigma \rangle$ approximates $\langle E', \Sigma' \rangle$,” or conversely, “ $\langle E', \Sigma' \rangle$ instantiates $\langle E, \Sigma \rangle$,” where $\langle E', \Sigma' \rangle$ is a concrete state and $\langle E, \Sigma \rangle$ is an abstract state. We make two remarks about the relation before defining it.

First, as discussed in section 3.2, when analyzing an incomplete program, we are only concerned with errors coming from known code. Therefore, we parameterize the approximation relation with a set of labels \vec{L} denoting application sites from the known program portion. Figure 6 presents the straightforward definition of metafunction *lab* for computing a program’s labels identifying application sites. The function takes a heap to compute labels for intermediate states, where a function may be referenced indirectly through its location. For the purpose of analyzing program E , the set of labels is $lab_{\Sigma}[[E]]$. As an example, expression E below has an instantiation E' , but when analyzing E , we are only interested in the potential division error at L and not L' .

$$\begin{aligned} E &= (\text{div } 1 \ (\bullet^{\text{int} \rightarrow \text{int}} 1))^L \\ E' &= (\text{div } 1 \ (\lambda x. (\text{div } 1 \ x)^{L'} 1))^L \end{aligned}$$

Second, we enforce that each location in the abstract state unambiguously approximates one location in the concrete state by parameterizing the approximation relation with a function F mapping each label in the abstract state to one in the concrete state. For example, we do not want the following concrete state $\langle E', \Sigma' \rangle$ to instantiate the abstract state $\langle E, \Sigma \rangle$, even though \bullet^{int} intuitively approximates each number 1, 2, and 3 individually.

$$\begin{aligned} \langle E, \Sigma \rangle &= \langle (\text{if } L \ L \ L), \{L \mapsto \bullet^{\text{int}}\} \rangle \\ \langle E', \Sigma' \rangle &= \langle (\text{if } L_1 \ L_2 \ L_3), \{L_1 \mapsto 1, L_2 \mapsto 2, L_3 \mapsto 3\} \rangle \end{aligned}$$

Instead, the following concrete state $\langle E'', \Sigma'' \rangle$ properly instantiates $\langle E, \Sigma \rangle$ with function $F = \{L \mapsto L_1\}$:

$$\langle E'', \Sigma'' \rangle = \langle (\text{if } L_1 \ L_1 \ L_1), \{L_1 \mapsto 1\} \rangle$$

Figure 7 defines the approximation parameterized by label set \vec{L} and function F . We present important, non-structural rules for the approximation relation between heaps, values, and states. We omit displaying parameters when they are unimportant or can be inferred from context. We defer the full definition to the appendix of the extended version of this paper [Nguyễn and Van Horn 2015].

Rule *Heap-Ext* states that if a heap approximates a concrete heap, it approximates any extension of that concrete heap. This rule is necessary for ignoring irrelevant computations in instantiation of an opaque function. Next, rules *Heap-Int*, *Heap-Lam*, *Heap-Opq-1*, and *Heap-Opq-2* show straightforward extensions to the approximation when the heaps on both sides are extended. First, any concrete value of the right type instantiates the opaque value \bullet^T as long as the instantiating value does not contain source locations from the known program portion. Second, refining an abstract value with a predicate known to be satisfied by the concrete value preserves the

$$\begin{aligned} lab_{\Sigma}[(O \ E)^L] &= \{L\} \cup lab_{\Sigma}[[E]] \\ lab_{\Sigma}[[E_1 \ E_2]] &= lab_{\Sigma}[[E_1]] \cup lab_{\Sigma}[[E_2]] \\ lab_{\Sigma}[[\text{if } E \ E_1 \ E_2]] &= lab_{\Sigma}[[E]] \cup lab_{\Sigma}[[E_1]] \cup lab_{\Sigma}[[E_2]] \\ lab_{\Sigma}[[\lambda X. E]] &= lab_{\Sigma}[[E]] \\ lab_{\Sigma}[[L]] &= lab_{\Sigma}[[\Sigma(L)]] \\ lab_{\Sigma}[[_]] &= \emptyset \end{aligned}$$

Figure 6. Computing concrete labels

approximation relation. Because a predicate can contain locations, we substitute labels appropriately as indicated by function F . We omit the straightforward definition of this substitution.

Rules *Heap-Case-1* and *Heap-Case-2* establish the approximation between functions on natural numbers. First, a fully opaque mapping approximates all functions. In addition, if there exists an execution trace witnessing that applying the concrete function yields a value approximated by an opaque value, then refining the mapping preserves the approximation.

Rule *Loc* states that location L approximates L' if the pair agrees with function F .

Rule *Err-Opq* reflects our decision of ignoring errors blaming source locations from unknown code. Otherwise, rule *Err* says that an error with a known label approximates another when they are the same error.

Finally, rule *Opq-App* states that we ignore irrelevant computation from a concrete function that instantiates an unknown function. Specifically, if we can establish that an opaque application approximates a concrete application by the structural rule, then the opaque application continues to approximate each non-answer state reachable from the concrete application. There are similar rules for approximation by applying other forms of opaque functions, which we defer to the appendix of the extended version of this paper [Nguyễn and Van Horn 2015].

Theorem 1 states that every constructed counterexample from an error case actually reproduces the same error. Notice that the theorem is conditioned on $\Sigma' \sqsubseteq \Sigma_2$ and does not imply that all errors in the abstract execution are real. In particular, if a path is spurious, its final heap has no instantiation.

Theorem 1 (Soundness of Counterexamples).

If $\langle E, \Sigma_1 \rangle \mapsto^* \langle \text{err}_O^L, \Sigma_2 \rangle$ and $\Sigma' \sqsubseteq \Sigma_2$, then $\langle E, \Sigma' \rangle \mapsto^* \langle \text{err}_O^L, \Sigma'' \rangle$.

Theorem 2 states that we can discover every potential bug and construct a counterexample for it, assuming the underlying solver is complete for queries on first-order data.

Theorem 2 (Relative Completeness of Counterexamples).

If $\langle E', \Sigma'_1 \rangle \mapsto^* \langle \text{err}_O^L, \Sigma'_2 \rangle$ and $\langle E', \Sigma'_1 \rangle \sqsubseteq_{\vec{L}} \langle E, \Sigma_1 \rangle$ and $L \in \vec{L}$, then $\langle E', \Sigma_1 \rangle \mapsto^* \langle \text{err}_O^L, \Sigma_2 \rangle$ such that there is an instantiation Σ' to Σ_2 .

4. Extensions

We discuss important extensions to our system for a more practical programming language with dynamic typing, data structures, contracts, and mutable states. In addition, we address the issue with termination. Our end goal is apply the method to realistic Racket [Flatt and PLT 2010] programs.

4.1 Dynamic typing

Dynamically typed languages defer safety checks to run-time to avoid conservative rejection of correct programs. Such languages have mechanisms for run-time inspection of data’s type tag. We model this feature by extending primitive predicates with run-time type tests such as `integer?` or `procedure?`, which operate in the

$$\begin{array}{c}
\text{Heap-Empty} \quad \frac{}{\emptyset \sqsubseteq_{\vec{L}}^{\{\}} \emptyset} \quad \text{Heap-Ext} \quad \frac{\Sigma' \sqsubseteq_{\vec{L}}^F \Sigma}{\Sigma'[L' \mapsto S'] \sqsubseteq_{\vec{L}}^F \Sigma} \\
\\
\text{Heap-Int} \quad \frac{\Sigma' \sqsubseteq_{\vec{L}}^F \Sigma}{\Sigma'[L' \mapsto n] \sqsubseteq_{\vec{L}}^{F[L' \mapsto L']} \Sigma[L \mapsto n]} \\
\\
\text{Heap-Lam} \quad \frac{\Sigma' \sqsubseteq_{\vec{L}}^F \Sigma \quad \langle E', \Sigma' \rangle \sqsubseteq_{\vec{L}}^F \langle E, \Sigma \rangle}{\Sigma'[L' \mapsto \lambda X.E'] \sqsubseteq_{\vec{L}}^{F[L' \mapsto L']} \Sigma[L \mapsto \lambda X.E]} \\
\\
\text{Heap-Opq-1} \quad \frac{\Sigma' \sqsubseteq_{\vec{L}}^F \Sigma \quad \text{lab}_{\Sigma'}[V'] \cap \vec{L} = \emptyset}{\Sigma'[L' \mapsto V'] \sqsubseteq_{\vec{L}}^{F[L' \mapsto L']} \Sigma[L \mapsto \bullet^T]} \\
\\
\text{Heap-Opq-2} \quad \frac{\Sigma'[L' \mapsto V'] \sqsubseteq_{\vec{L}}^F \Sigma[L \mapsto \bullet^{TP\dots}] \quad \Sigma' \vdash V' : F(P_1) \checkmark}{\Sigma'[L' \mapsto V'] \sqsubseteq_{\vec{L}}^F \Sigma[L \mapsto \bullet^{TP\dots P_1}]} \\
\\
\text{Heap-Case-1} \quad \frac{\Sigma' \sqsubseteq_{\vec{L}}^F \Sigma \quad \text{lab}_{\Sigma'}[E'] \cap \vec{L} = \emptyset}{\Sigma'[L' \mapsto \lambda X.E'] \sqsubseteq_{\vec{L}}^{F[L' \mapsto L']} \Sigma[L \mapsto \text{case}^T []]} \\
\\
\text{Heap-Case-2} \quad \frac{\Sigma''[L' \mapsto \lambda X.E'] \sqsubseteq_{\vec{L}}^F \Sigma[L \mapsto \text{case}^T [\dots]] \quad F(L_x) = L'_x \quad \langle [X/L'_x]E', \Sigma'' \rangle \mapsto^* \langle V', \Sigma' \rangle \quad \langle V', \Sigma' \rangle \sqsubseteq_{\vec{L}}^F \langle V, \Sigma \rangle}{\Sigma'[L' \mapsto \lambda X.E'] \sqsubseteq_{\vec{L}}^{F[L' \mapsto L']} \Sigma[L \mapsto \text{case}^T [\dots L_x \mapsto V]]} \\
\\
\text{Loc} \quad \frac{F(L) = L'}{\langle L', \Sigma' \rangle \sqsubseteq_{\vec{L}}^F \langle L, \Sigma \rangle} \quad \text{Err-Opq} \quad \frac{L' \notin \vec{L}}{\langle \text{err}_{\vec{L}}^{L'}, \Sigma' \rangle \sqsubseteq_{\vec{L}}^F \langle E, \Sigma \rangle} \\
\\
\text{Err} \quad \frac{L' \in \vec{L}}{\langle \text{err}_{\vec{L}}^{L'}, \Sigma' \rangle \sqsubseteq_{\vec{L}}^F \langle \text{err}_{\vec{L}}^{L'}, \Sigma \rangle} \\
\\
\text{Opq-App} \quad \frac{E' \neq A \quad \text{lab}_{\Sigma'}[E''] \cap \vec{L} = \emptyset \quad \langle L'_f L'_x, \Sigma'' \rangle \sqsubseteq_{\vec{L}}^F \langle L_f L_x, \Sigma \rangle \quad \Sigma''(L'_f) = \lambda X.E'' \quad \Sigma(L_f) = \bullet^{T \rightarrow T'} \quad \langle [X/L'_x]E'', \Sigma'' \rangle \mapsto^* \langle E', \Sigma' \rangle}{\langle E', \Sigma' \rangle \sqsubseteq_{\vec{L}}^F \langle (L_f L_x), \Sigma \rangle}
\end{array}$$

Figure 7. Approximation

same manner as zero? in the typed language. Changes to the semantics are straightforward: we insert a run-time check into each application to ensure a function is begin applied, and into each primitive application to ensure arguments have the right tags. We also modify the rules for applying unknown functions, where previous static distinction in function types are turned to corresponding dynamic checks.

4.2 User-defined data structures

We extend the semantics to allow user-defined data structures, enabling programmers to express rich data such as lists and trees. Below is an example definition of a binary tree's node:

```
(struct node (left content right))
```

Each field in a data structure may itself be another data structure, function, or base value. Following the same treatment as functions, we do not encode data structures in the solver. Instead, we rely on execution to incrementally refine an unknown value's shape when knowing that it has a specific tag. For example, an unknown node has the shape of $(\text{node } L_1 L_2 L_3)$ where each of the fields L_1 , L_2 and L_3 is an unknown and refinable value. As before, we only need to encode constraints on base values at the leaves of data structures.

4.3 Contracts

Contracts generalize pre-and-post conditions to higher-order specifications [Findler and Felleisen 2002], allowing programmers to express rich invariants using arbitrary code. They can either refine an existing type system [Hinze et al. 2006] or ensure safety in an untyped language.

The following Racket [Flatt and PLT 2010] program illustrates the use of a higher-order contract. Function `argmin` requires a number-producing function as its first argument and a list as its second, and returns the list's element that minimizes the function.

```
; (argmin f xs) → any/c
; f : (any/c → number?)
; xs : (and/c pair? list?)
(define (argmin f xs)
  (argmin/acc f (car xs) (f (car xs)) (cdr xs)))

(define (argmin/acc f b a xs)
  (cond
    [(null? xs) a]
    [(< b (f (car xs))) (argmin/acc f a b (cdr xs))]
    [else (argmin/acc f (car xs) (f (car xs)) (cdr xs))]))
```

Although the semantics of contract checking can be complex [Greenberg et al. 2010; Dimoulas et al. 2011], it introduces no new challenges in our system. We simply rely on the semantics of contract checking itself to break down complex and higher-order contracts into simple predicates. In addition, opaque flat contracts can be modeled soundly and precisely by rules for opaque application. Extension to the contract checking semantics enables our system to construct counterexamples to violated higher-order contracts.

4.4 Mutable state

We support stateful programs by extending the language with primitives for assigning to and dereferencing mutable cells, along with a type tag predicate.

When an unknown function is applied to a mutable cell, it may invoke its content and mutate the cell arbitrarily. Second, if its argument is a function, the unknown context may apply the function any number of times, affecting the argument's internal state arbitrarily. Finally, in the presence of mutable states, the system can no longer assume that each function yields equal outputs for equal inputs, so a memoized mapping is no longer applicable. Because this last change can be too conservative for reasoning about idiomatic

functional programs, where programmers often think of functions as pure and use mutable cells judiciously, it is useful in a practical system to have a special annotation for marking an unknown function as pure.³

One challenge introduced by mutable cells is aliasing. For example, a result from applying an unknown function can either be a fresh value, or any previous value on the heap. Future side effects performed on this unknown result may or may not affect an existing mutable cell. To soundly execute symbolic programs with mutable states, we modify the behavior of primitive `box?` for run-time testing of mutable cells. When an unknown value L is determined to be a mutable cell, it is non-deterministically a distinct cell from any previous one on the heap or an alias to each previous cell and perform a substitution in the entire program. Although this process is expensive, mutable cells are sparse in idiomatic Racket programs. Programs with no invocation of `box?` (which is implicit in other operations) do not pay this cost. More efficient handling of aliasing in large imperative programs is one direction of our future work.

4.5 Termination

The semantics presented so far does not guarantee termination. We can either accelerate (but not guarantee) termination by detecting recursion and widen values accordingly [Nguyễn et al. 2014], or guarantee termination through systematic transformation of the semantics into a finite state or pushdown analysis of itself [Van Horn and Might 2010]. These techniques introduce spurious paths as over-approximations to actual execution branches. This affects both soundness and completeness of counterexamples. First, it requires more work to guarantee soundness. Because multiple concrete traces may be approximated to the same abstract trace, running the program with one instantiation of a constraint set may steer the program’s flow to a different concrete trace that has the same abstraction. To ensure an instantiation corresponds to a real counterexample, it is necessary to first run the program with the concrete value set before reporting it as a counterexample. Second, relative completeness is also lost in practice. Even though execution still reveals every possible error, approximation results in a less precise constraint set for each trace, and the system may repeatedly query the solver for the wrong model before timing out. For example, a simplistic solver trying to refute that “ $\text{factorial}(n + 4) \geq 10$ ” with no constraint may keep producing non-negative values for n .

Nevertheless, for our specific need of counterexample generation to refine an existing verification system (discussed next in section 5), we perform no abstraction. We rely on the previous system to prove the lack of counterexamples for a large set of correct programs [Nguyễn et al. 2014] (therefore, many correct programs without counterexamples do terminate). When used in combination with a verification system, abstracting the state space for counterexample generation is of little value, and makes it difficult to later concretize values to obtain a counterexample.

5. Implementation and evaluation

To evaluate our approach, we integrate counterexample generation into an existing contract verification system for programs written in a subset of Racket [Flatt and PLT 2010]. The system previously either successfully verified correct programs or conservatively reported probable contract violations and did not distinguish definite program errors from potentially false positives. With the new enhancement, the tool identifies a subset of reported errors as definite bugs with concrete counterexamples. Below, we describe implementation extensions, discuss promising experiment results, and address current difficulties.

5.1 Implementation

Our implementation and benchmarks can be found at

github.com/philnguyen/soft-contract/tree/pldi-2015

The prototype handles a much more realistic set of language features beyond SPCF. First, our implementation supports dynamic typing with user-defined structures and first class contracts as discussed in section 4. We also support more contract combinators such as conjunction, disjunction, and recursion. Second, we extend the set of base values and primitive operations, such as pairs, strings and Racket’s full numeric tower, which introduces more error sources and interesting counterexamples. Finally, we employ a module system to let users organize code. A module can export multiple values and define private ones for internal use.

Apart from being implemented as a command line tool, our prototype is also available as a web REPL at

scv.umiacs.umd.edu

The system attempts to verify correct programs and refute erroneous programs with concrete counterexamples. In some cases, it reports a probable contract violation without giving any counterexample due to limitations of the underlying solver, or the server simply times out after 10 seconds.

5.2 Evaluation

We collect benchmarks for our analysis from two sources: (1) prior published work and (2) submissions to the web REPL we built.

Benchmarks from prior work are drawn from research on higher-order model checking [Kobayashi et al. 2011], dependent type checking [Terauchi 2010], occurrence type checking [Tobin-Hochstadt and Felleisen 2010], and our own work on contract verification [Nguyễn et al. 2014]. Since these prior works focus on verification, the benchmarks are largely correct programs. In order to evaluate our counterexample generation technique, we modify each of the programs to introduce errors. To do so, we weakened preconditions and omitted checks before performing partial operations. For example, a resulting program may deconstruct a potentially empty list or compare potentially non-real numbers. We believe these changes are representative of common mistakes. A complete listing of the modifications is available.⁴

Benchmarks from our web service are submitted (anonymously) by users experimenting with the verification system. Many of these programs are buggy and we test how effective at discovering counterexamples.

In total, the evaluation is run on 282 programs consisting of 4050 lines of Racket code, excluding empty lines and comments. The largest programs are three student video games of the order of 250 lines. The test suite includes correct programs the system tries to verify as well as incorrect programs the system tries to generate counterexamples for.

We summarize our benchmark results in table 1. Each row shows the program’s size (column 1), its highest function order (column 2), the time taken to verify the correct version of the program (column 3, if applicable), and the time taken to generate a counterexample refuting an incorrect variant of the program (column 4, if applicable). We compute each program’s order by inspecting its contract’s syntax (which is an under-approximation, because a contract may be dynamically computed). The last 3 rows “others”, “others-e” and “others-w” summarize many small programs from our own benchmark suite as well as those collected from the server; we report their total, minimum and maximum line counts, total verification time, and highest function orders. With the exception of 5 programs in the last row “others-w”, the system gives a counterex-

³First-class contracts can have internal states and enforce extensionality, which symbolic execution can make use of.

⁴<https://github.com/philnguyen/soft-contract/blob/pldi-2015/soft-contract/benchmark-verification/diff.txt>

ample for each incorrect program in a reasonable amount of time: the most complicated error takes 7 seconds to detect, and most errors in typical higher-order programs take less than 2 seconds. The last row shows benchmarks (all contributed by anonymous users) that reveal the limitation of our counterexample generation in practice. In each of these cases, the system soundly reports a probable contract violation, but is unable to generate a counterexample confirming it. We discuss current shortcomings and language features known to thwart the tool in section 5.3.

The overall result is promising. First, there are specific examples where our prototype proves to be a good complement to random testing. For example, the tool finds a counterexample to the following program quickly and automatically:

```
f n = (/ 1 (- 100 n))
```

By default, QuickCheck does not find this error as it only considers integers from -99 to 99. Because QuickCheck treats a program as a black box, this conservative choice is reasonable for fear that the integer may be a loop variable causing the test case to run for a long time [Hughes 2015]. In contrast, our method explores the program’s semantics symbolically and discovers 100 as a good test case.

Second, the resulting higher-order counterexamples suggest that the analysis can produce useful feedback. For example, it is easy for programmers to forget that Racket supports the full numeric tower [St-Amour et al. 2012] and that the predicate `number?` accepts complex numbers. The contract on `argmin` in section 4.3 is in fact too weak to protect the function. The system proves `argmin` unsafe by applying it to a specific combination of arguments. First, `f` is given a function that produces a non-real number, which causes `<` to signal an error. Second, `xs` is given a list of length 2, which is the minimum length to trigger a use of `<`.

```
f = (λ (x) 0+1i); xs = (list 0 0)
```

Finally, the tool analyzes the functional encoding of object-oriented programs effectively. Zombie is one such example with extensive use of higher-order functions to encode objects and classes, and the analysis can reveal errors buried in delayed function calls. We believe this is a promising first step for generating classes and objects as counterexamples. In the example below, we define interface `posn/c` that accepts two messages `x` and `y`, and function `first-quadrant?` that tests whether a position is in the first quadrant.

```
(define posn/c
  ([msg : (one-of/c 'x 'y)])
  → (match msg ['x number?] ['y number?]))

; posn/c → boolean?
(define (first-quadrant? p)
  (and (≥ (p 'x) 0) (≥ (p 'y) 0)))
```

The counterexample reveals one conforming implementation to interface `posn/c` that causes error in the module.

```
(λ (msg) (case msg [(x) 0+1i] [(y) 0]))
```

5.3 Difficulties

We discuss current difficulties to our approach and solutions in mitigating them.

First, the analysis is prone to combinatorial explosion as inherent in symbolic execution. Our tool finds bugs by performing a simple breadth-first search on the execution graph, then stops and reports on the first error encountered with a fully concrete counterexample. In practice, most conditionals come from case analyses instead of independent alternatives, and we rely on a precise proof system to eliminate spurious paths. In addition, the modularity mitigates the problem further, as modules tend to be small, and contracts at boundaries help recovering necessary precision.

Program	Lines	Order	Correct (ms)	Incorrect (ms)
Kobayashi et al. 2011 benchmarks				
fhnhn	18	2	38	50
fold-div	18	2	321	160
fold-fun-list	20	3	92	442
hors	25	2	49	34
hrec	9	2	52	143
intro1	13	2	24	128
intro2	13	2	25	127
intro3	13	2	25	23
isnil	9	1	13	9
max	14	2	32	135
mem	12	1	22	254
mult	9	1	61	147
nth0	15	1	19	296
r-file	50	1	74	123
r-lock	17	1	56	49
reverse	11	1	15	205
Terauchi 2010 benchmarks				
boolflip	10	1	10	22
mult-all	10	1	9	225
mult-cps	12	1	253	35
mult	10	1	72	21
sum-acm	10	1	33	833
sum-all	9	1	8	186
sum	8	1	44	19
Tobin-Hochstadt and Felleisen 2010 benchmarks				
occurrence (14)	116	1	99	226
Nguyễn et al. 2014 benchmarks (video games)				
snake	164	1	37,350	2,476
tetris	267	2	11,809	2,188
zombie	249	3	19,239	954
Nguyễn et al. 2014 other benchmarks and anonymous web submissions				
others (73)	(2 - 51) 818	3	20,465	-
others-e (124)	(3 - 23) 972	3	-	19,588
others-w (5)	(4 - 4) 20	1	-	431*

Table 1. Program verification and refutation time

One major source of slowdown in our system is complex preconditions, where each input is guarded against a deep, inductively defined property. Execution follows different branches before being able to generate a valid input to continue verifying the module. A naive breadth-first search is bogged down by a large frontier resulting from different attempts to generate inputs, most of which are eventually found invalid. To mitigate this slow-down, we identify a class of expressions as likely to lead to counterexamples and prioritize their execution. Specifically, an expression whose innermost contract monitoring is of a first-order property on a concrete module is likely to reveal a bug.⁵ In contrast, expressions in the middle of input generation do not have this form, because the inner-most contract monitoring is on the opaque input source. Once the system successfully instantiates a concrete input and turns the program into this “suspect” form, it focuses on exploring this branch with that input instead of trying numerous other inputs in parallel. Using this simple heuristic, we are able to cut the execution time of a module violating the “braun-tree” invariant from non-terminating after 1 hour down to 2 seconds.

Second, there is a mismatch in the data-types between Z3’s data-type and Racket’s rich numeric tower. In particular, Racket supports mixed arithmetic between different types of numbers up to complex

⁵In a symbolic program, the monitored value in this position is usually abstract and covers all values the module produces.

numbers [St-Amour et al. 2012], while Z3’s treatment of numbers resembles that from most statically typed languages, and the solver does not perform well in generating models involving a dynamic restriction of a number’s type. Below is an example in the last row in table 1 where the tool fails to generate a counterexample:

```
; (integer? → integer?)
(define (f n) (/ 1 (+ 1 (* n n))))
```

In Racket, division is defined on the full numeric tower, and the result of `(/ 1 (+ 1 (* n n)))` may not be an integer. In the generated query, this result is an unknown number `L` of type `Real`, and the solver cannot give a model to a constraint set asserting “(not (is_int L))”. In addition, Racket distinguishes between exact and inexact numbers, where inexact numbers are floating point approximations. Because Z3 does not reason about floating points, we currently do not soundly model inexact arithmetic.

6. Related work

We relate our work to four main lines of research: symbolic execution, counterexample guided abstraction refinement for dependent type inference, random testing, and contract verification.

First-order Symbolic Execution Symbolic execution on first-order programs is mature and has been used to find bugs in real-world programs [Cadaru et al. 2006, 2008]. Cadaru et al. [2006] presents a symbolic execution engine for C that generates counterexamples of the form of mappings from addresses to bit-vectors. Later work extends the technique to generate comprehensive test cases that discover bugs in large programs interacting with the environment [Cadaru et al. 2008].

Counterexample-guided Abstraction Refinement CEGAR has been used in model checking and dependent type inference [Rondon et al. 2008; Kobayashi et al. 2011; Zhu and Jagannathan 2013], where the inference algorithm iteratively uses a counterexample given by the solver to refine preconditions attached to functions and values. In case the algorithm fails to infer a specification, the counterexample serves as a witness to a breaking input. Our work finds higher-order counterexamples only by integrating a first-order solver, and is applicable to both typed and untyped languages. In contrast, dependent type inference relies on an extension to ML. In addition, work on higher-order model checking analyzes complete programs with first-order unknown inputs, while we analyze partial programs with potentially higher-order unknown values at the boundaries.

Random Testing Random testing is a lightweight technique for finding counterexamples to program specifications through randomly generated inputs. QuickCheck for Haskell [Claessen and Hughes 2000] proves the approach highly practical in finding bugs for functional programs. Later works extend random testing to improve code coverage and scale the technique to more language features such as states and class systems. Heidegger and Thiemann [2010] use contracts to guide random testing for Javascript, allowing users to annotate inputs to combine different analyses for increasing the probability of hitting branches with highly constrained preconditions. Klein et al. [2010] also extend random testing to work on higher-order stateful programs, discovering many bugs in object-oriented programs in Racket. Seidel et al. [2015] use refinement types as generators for tests, significantly improving code coverage.

Our approach is a complement to random testing. By combining symbolic execution with an SMT solver, the method takes advantage of conditions generated by ordinary program code and not just user-annotated contracts. In addition, the approach works well with highly constrained preconditions without further help from users. In contrast, random testing systems typically require programmers

to implement custom generators [Claessen and Hughes 2000] or require user annotations to incorporate a specific analysis collecting all literals in the program to guide input construction [Heidegger and Thiemann 2010]. Type-targeted testing [Seidel et al. 2015] is more lightweight and does not necessitate an extension to the existing semantics, but gives no guarantee about completeness, as inherent in random testing. Even though the tool rules out test cases that fail the pre-conditions, regular code and post-conditions do not help the test generation process. Our system makes use of both contracts and regular code to guide the execution to seek inputs that both satisfy pre-conditions and fail post-conditions. Exploring possible combination of symbolic execution and random testing for more efficient bug-finding in higher-order programs is our future work.

Contract Verification and Refinement Type Checking Contracts and refinement types are mechanisms for specifying much richer program invariants than those allowed in a typical type system. Verification systems either restrict the language of refinements to be decidable [Rondon et al. 2008] or allow arbitrary enforcement but leave unverifiable invariants as residual run-time checks [Flanagan 2006; Knowles and Flanagan 2010; Xu 2012; Tobin-Hochstadt and Van Horn 2012]. While verification proves the absence of errors but may give false positives, our tool aims to discover concrete, real counterexamples to faulty programs. Our work is a direct extension to previous work on symbolic execution of higher-order programs [Tobin-Hochstadt and Van Horn 2012] and can be viewed as a complement to contract verification.

7. Conclusion

We have presented a symbolic execution semantics for finding concrete counterexamples in higher-order programs and proved it to be sound and relatively complete. An early prototype shows that the approach can scale to realistically sized functional programs with practical features such as first-class contracts. From the programmer’s perspective, the approach is lightweight and requires no custom annotation to get started. However, if contracts are present, they can help guide the search for counterexamples. Combined with previous work on contract verification, it is possible to construct a tool that can statically guarantee contract correctness of programs and simultaneously ease the understanding of faulty programs, speeding up the development of reliable software.

Acknowledgments

We thank Sam Tobin-Hochstadt for countless discussions that contributed significantly to the development of this work. Robby Findler provided considerable feedback that improved the presentation of this paper. Clayton Mentzer helped build the web REPL and Andrew Reuf gave valuable guidance on our artifact submission. We thank John Hughes and Suresh Jagannathan for comments on prior work. We thank the PLDI and PLDI ERC reviewers for their detailed reviews, which helped to improve the presentation and technical content of the paper and accompanying artifact. This research is supported in part by the National Security Agency under the Science of Security program.

References

- C. Cadaru, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*. ACM, 2006.
- C. Cadaru, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. USENIX, 2008.

- K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ACM, 2000.
- S. A. Cook. Soundness and completeness of an axiom system for program verification. In *SIAM Journal of Computing*, 1978.
- L. De Moura and N. Björner. Z3: an efficient SMT solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*. Springer-Verlag, 2008.
- C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. Correct blame for contracts: no more scapegoating. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2011.
- C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen. Complete monitors for behavioral contracts. In *21st European Symposium on Programming*. Springer Berlin Heidelberg, 2012.
- R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming*. ACM, 2002.
- C. Flanagan. Hybrid type checking. In *Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2006.
- M. Flatt and PLT. Reference: Racket. Technical report, PLT Inc., 2010.
- J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2002.
- P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2005.
- M. Greenberg, B. C. Pierce, and S. Weirich. Contracts made manifest. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2010.
- P. Heidegger and P. Thiemann. Contract-Driven testing of JavaScript code. In *Objects, Models, Components, Patterns*. Springer Berlin Heidelberg, 2010.
- R. Hinze, J. Jeuring, and A. Löb. Typed contracts for functional programming. In *Functional and Logic Programming*. Springer, 2006.
- J. Hughes. Personal communication, 2015.
- M. Kawaguchi, P. Rondon, and R. Jhala. Dsolve: Safety verification via liquid types. In *Computer Aided Verification*. Springer Berlin Heidelberg, 2010.
- C. Klein, M. Flatt, and R. B. Findler. Random testing for higher-order, stateful programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, 2010.
- K. Knowles and C. Flanagan. Hybrid type checking. *ACM Trans. Program. Lang. Syst.*, 2010.
- N. Kobayashi. Model checking Higher-Order programs. *J. ACM*, 2013.
- N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and CEGAR for higher-order model checking. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2011.
- P. C. Nguyễn and D. Van Horn. Relatively complete counterexamples for Higher-Order programs. *CoRR*, abs/1411.3967, 2015.
- P. C. Nguyễn, S. Tobin-Hochstadt, and D. Van Horn. Soft contract verification. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ACM, 2014.
- P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2008.
- D. S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 1993.
- E. L. Seidel, N. Vazou, and R. Jhala. Type targeted testing. In *21st European Symposium on Programming*. Springer Berlin Heidelberg, 2015.
- V. St-Amour, S. Tobin-Hochstadt, M. Flatt, and M. Felleisen. Typing the numeric tower. In *Practical Aspects of Declarative Languages*. Springer Berlin Heidelberg, 2012.
- T. Terauchi. Dependent types from counterexamples. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2010.
- S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. In *Proceedings of the ACM International Conference on Functional Programming*. ACM, 2010.
- S. Tobin-Hochstadt and D. Van Horn. Higher-order symbolic execution via contracts. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, 2012.
- D. Van Horn and M. Might. Abstracting abstract machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ACM, 2010.
- D. Vytiniotis, S. Peyton Jones, K. Claessen, and D. Rosén. HALO: Haskell to logic through denotational semantics. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 2013.
- Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2005.
- D. N. Xu. Hybrid contract checking via symbolic simplification. In *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation*. ACM, 2012.
- D. N. Xu, S. Peyton Jones, and S. Claessen. Static contract checking for Haskell. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2009.
- J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Sixth Symposium on Operating Systems Design and Implementation*. USENIX, 2004.
- H. Zhu and S. Jagannathan. Compositional and lightweight dependent type inference for ML. In *Conference on Verification, Model-Checking and Abstract Interpretation*, 2013.